(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
9 October 2003 (09.10.2003)

PCT

(10) International Publication Number
WO 03/083614 A2

(51) International Patent Classification[7]: G06F

(21) International Application Number: PCT/US03/09233

(22) International Filing Date: 24 March 2003 (24.03.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/367,615 25 March 2002 (25.03.2002) US
60/367,616 25 March 2002 (25.03.2002) US

(71) Applicant *(for all designated States except US)*: **ETERNAL SYSTEMS, INC.** [US/US]; 1901 South Bascom Avenue, Suite 1200, San Jose, CA 95008 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **MOSER, Louise, E.** [US/US]; P.O. Box 13963, Santa Barbara, CA 93107 (US). **MELLIAR-SMITH, Peter, M.** [US/US]; P.O. Box 13963, Santa Barbara, CA 93111 (US).

(74) Agent: **O'BANION, John, P.**; O'Banion & Ritchey LLP, 400 Capitol Mall, Suite 1550, Sacramento, CA 95814 (US).

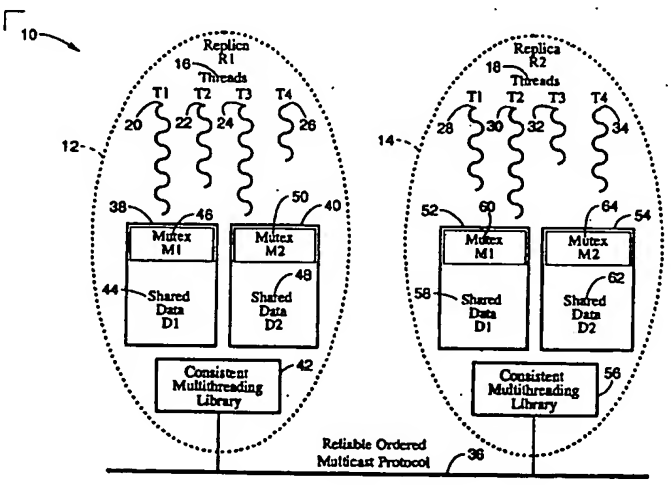
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NI, NO, NZ, OM, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, RO, SE, SI, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: TRANSPARENT CONSISTENT ACTIVE REPLICATION OF MULTITHREADED APPLICATION PROGRAMS



(57) Abstract: A method and system for transparent consistent active replication of multithreaded application programs is described. At each replica, control messages that contain mutex ordering information indicating the order in which threads in the replicas claim mutexes are multicast, and the control messages are delivered using a multicast group communication protocol that delivers the messages in an order that determines the order in which the operating system's thread library grants the claims of mutexes to the threads in the replicas. Because the replicas receive the same messages in the same source order, the corresponding threads in the different replicas are granted their corresponding claims to the corresponding mutexes in the same order, maintaining strong replica consistency.

WO 03/083614 A2

TITLE OF THE INVENTION

TRANSPARENT CONSISTENT ACTIVE REPLICATION OF MULTITHREADED APPLICATION PROGRAMS

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims priority from U.S. provisional application serial number 60/367,615 filed on March 25, 2002, incorporated herein by reference, and from U.S. provisional application serial number 60/367,616 filed on March 25, 2002, incorporated herein by reference.

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

[0002] This invention was made with Government support under Grant No. 70NANBOH3015, awarded by the U.S. Department of Commerce, National Institute of Standards and Technology. The Government may have certain rights in this invention.

INCORPORATION-BY-REFERENCE OF MATERIAL SUBMITTED ON A COMPACT DISC

[0003] Not Applicable

NOTICE OF MATERIAL SUBJECT TO COPYRIGHT PROTECTION

BACKGROUND OF THE INVENTION

1. Field of the Invention

[0005] This invention pertains generally to software-based fault-tolerant computer systems and, more particularly, to multithreaded application programs that are replicated using the egalitarian and competitive active replication strategy.

2. Description of Related Art

[0006] Fault-tolerant systems are based on entity redundancy (replication) to mask faults and, thus, to provide continuous service to their users. In software fault tolerance, the entities that are replicated are the application programs or parts thereof (processes, objects or components). A fundamental issue in the design and implementation of fault-tolerant systems is that of maintaining consistency of the states of the replicas.

[0007] Distributed systems offer the opportunity for fault tolerance by allowing replicas of the application programs to be hosted on different computers (i.e., in different fault containment regions). In the client-server model of distributed computing, a client invokes a method of a server, typically hosted on a different computer. To render an application fault-tolerant, the server is replicated but the client may also be replicated, particularly in multi-tier applications and in peer-to-peer applications, wherein a process, object or component acts as both a client and a server.

[0008] In an *active replication* strategy, the program code of the replicas is identical and the replicas execute their copies of the code concurrently and, thus, the active replication strategy is an egalitarian strategy. Active replication is based on each of the replicas starting in the same initial state (values of their attributes or variables) and executing the same methods or operations and on strong replica consistency. If there is no non-determinism in the execution of the replicas, it is obvious that they will reach the same state at the end of the execution of each method invocation or operation. The present invention ensures that the replicas generate the same results, even if non-determinism caused by multi-threading is present in the replicas. When the replicas take an action or produce a result that is externally visible, such

as sending a message, issuing an input/output command, and so forth, the first such action or result is the one that is used and the corresponding actions or results of the other replicas are either suppressed or discarded. Thus, the active replication strategy is a competitive strategy.

[0009] The most challenging aspect of replication is maintaining strong replica consistency, as methods are invoked on the replicas, as the states of the replicas change dynamically, and as faults occur. *Strong replica consistency* means that, for each method invocation or operation, for each data access within said method invocation or operation, the replicas obtain the same values for the data. Moreover, for each result, message sent or request made to other processes, objects or components, the replicas generate the same result, message or request.

[0010] Many application programs written in modern programming languages (such as C++, Java, etc.) involve multithreading, which is a source of non-determinism. Unless it is properly handled, non-determinism can lead to inconsistency in the states of the replicas. To maintain strong replica consistency, it is necessary to sanitize or mask such sources of non-determinism, i.e., to render the replicated application program virtually deterministic. A *virtually deterministic* replicated application program is an application program that exists as two or more replicas and that may involve non-deterministic decisions but, for those non-deterministic decisions that affect the state of the replicas at the end of each method invocation, the replicas make the same non-deterministic decisions.

[0011] Many fault-tolerant systems based on active replication employ a *multicast group communication system*. Examples of such a multicast group communication system are Isis (K. P. Birman and R. van Rennesse, Reliable Distributed Computing Using the Isis Toolkit, IEEE Computer Society Press, 1994, incorporated herein by reference) and Totem (L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos, Totem: A fault-tolerant multicast group communication system, Communications of the ACM, vol. 39, no. 4, April 1996, pp. 54-63, incorporated herein by reference). Such a multicast group communication

system delivers messages reliably and in the same order (linear sequence) to all of the members of the group, i.e., to all of the replicas of the process, object or component.

[0012] For replicated unithreaded application programs where the replicas are distributed on multiple computers, a reliable ordered multicast group communication system can be used to maintain strong replica consistency, in the presence of no other sources of non-determinism except the order in which messages are delivered. For multithreaded application programs, the problem of maintaining strong replica consistency is more difficult because two threads in a replica can access a shared resource, typically shared data, in an order different from the order in which the corresponding threads in another replica access their copies of the shared data; consequently, the states of the replicas can diverge and become inconsistent.

[0013] For multithreaded application programs, if two threads within a process, object or component share data between them, only one of those threads can access that shared data at a time. Therefore, the shared data must be protected with a mutual exclusion construct, commonly referred to as a *mutex*, and the thread must be granted the mutex, and enter the *critical section* of code within which it can access the shared data. When the thread is finished accessing the shared data, it must release the mutex and leave the critical section. To maintain strong replica consistency, the threads in the replicas must be granted the mutexes in the same order, so that they enter the critical sections and access the shared data within the critical section in the same order.

[0014] There are several prior patents that address multithreaded application programs. In particular, U.S. Patents 5,577,261 and 5,794,043, which are incorporated herein by reference, describe the implementation of process management functions, such as the claim(), release(), suspend() and signal() functions. Operations involving those functions are rendered consistent by having each processor claim a global mutex (called GLUPP) before performing any process management operation. Once it has acquired the global mutex, the process performs the operation and then distributes the

results to the other processors before relinquishing the global mutex.

[0015] The global mutex, used in those patents, is actually described in U.S. Patent 4,718,002, which is incorporated herein by reference. That patent describes how a mutex can be granted to processors, processes, replicas or threads in a distributed system, but the mechanism requires that one processor should be designated as a distinguished control processor and that the granting of the mutex is determined by that control processor.

[0016] U.S. Patent 5,621,885, which is incorporated herein by reference, describes a strategy based on a Primary/Backup approach, in which the Primary replica executes the required operations. When the Primary replica performs an I/O operation, the results of the I/O operation are communicated to the Backup replica, so that the Backup replica performs the same operation as the Primary replica. That strategy requires the replicas to be cast into specific roles of either Primary or Backup replica.

[0017] U.S. Patents 5,802,265 and 5,968,185, which are incorporated herein by reference, likewise describe a strategy based on a Primary/Backup approach, in which the Primary replica executes the operations required of the computer system. When the Primary replica performs an asynchronous or non-deterministic interaction with the operating system, the results of the interaction with the operating system are communicated to the Backup replica, so that the Backup replica performs the same operation as the Primary replica. Object code editing is the primary mechanism by which the program code is modified and no provisions are made for active replication. U.S. Patents 5,802,265 and 5,968,185 are related to the TARGON/32 Fault Tolerance (TFT) system, described below.

[0018] The TARGON/32 system (A. Borg, W. Blau, W. Graetsch, F. Herrmann and W. And, Fault tolerance under Unix, ACM Transactions on Computer Systems, vol. 7, no. 1, 1989, pp. 1-24, incorporated herein by reference) describes a fault-tolerant version of the Unix operating system. It is based on special hardware that provides a reliable ordered multicast protocol, but is not applicable to distributed systems. Moreover, that strategy requires a distinguished control processor.

[0019] The Delta-4 system (M. Chereque, D. Powell, P. Reynier, J. L. Richier and J. Voiron, Active replication in Delta-4, Proceedings of the IEEE 22nd International Symposium on Fault Tolerant Computing, Boston, MA, 1992, pp. 28-37 and also D. Powell (ed.), Delta-4: A Generic Architecture for Dependable Distributed Computing, Springer-Verlag, 1991, both of which are incorporated herein by reference) supports active, semi-active and passive replication for application programs, but it does not handle non-determinism (in particular, multithreading) for active replication.

[0020] The Hypervisor system (T. C. Bressoud and F. B. Schneider, Hypervisor-based fault tolerance, ACM Transactions on Computer Systems, vol. 14, no. 1, 1996, pp. 80-107, incorporated herein by reference) and the Transparent Fault Tolerance (TFT) system (T. C. Bressoud, TFT: A software system for application-transparent fault tolerance, Proceedings of the IEEE 28th Fault-Tolerant Computing Symposium, Munich, Germany, June 1998, pp. 128-137, incorporated herein by reference) both aim for transparency to the application and the operating system. However, the Hypervisor system uses hardware instruction counters to count the instructions executed between two hardware interrupts and the TFT system uses object code editing to modify the program code. Moreover, both of those systems employ a Primary/Backup approach.

[0021] Other researchers (J. H. Sly and E. N. Elnozahy, Supporting non-deterministic execution in fault-tolerant systems, Proceedings of the IEEE 26th Fault Tolerant Computing Symposium, Sendai, Japan, June 1996, pp. 250-259, incorporated herein by reference) have introduced a software instruction counter approach, analogous to the hardware instruction counter approach of the Hypervisor system, to count the number of instructions between non-deterministic events in log-based rollback-recovery systems.

[0022] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, Enforcing determinism for the consistent replication of multithreaded CORBA applications, Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems, Lausanne, Switzerland, October 1999, pp. 263-273, incorporated herein by reference, describes a non-preemptive deterministic

scheduler strategy that imposes a single logical thread of control on the replicas to maintain strong replica consistency. That strategy, in effect, undoes the multithreading that was programmed into the application program.

[0023] Transactional Drago (S. Arevalo, R. Jimenez-Peris and M. Patino-Martinez, Deterministic scheduling for transactional multithreaded replicas, Proceedings of the IEEE 19th Symposium on Reliable Distributed Systems, Numberg, Germany, October 2000, pp. 164-173, incorporated herein by reference) also uses a non-preemptive deterministic scheduler but is aimed at transaction processing systems.

[0024] It will be appreciated that strategies, such as detailed in U.S. Patents 4,718,002, 5,621,885, 5,802,265 and 5,968,185 described above, require casting replicas into Primary or Backup roles. Furthermore, U.S. Patents 5,802,265 and 5,968,185 and the TFT system utilize object code editing to modify the program code and they disclose no mechanisms for active replication.

[0025] Therefore, a need exists for a system and method of providing consistent replication of multithreaded applications that is egalitarian and may be transparently implemented. The present invention satisfies those needs, as well as others, and overcomes the deficiencies of previously developed active replication strategies.

BRIEF SUMMARY OF THE INVENTION

[0026] The mechanisms of this invention aim to achieve strong replica consistency of multithreaded application programs that are replicated using the egalitarian and competitive active replication strategy. This invention is applicable to distributed systems in which the several computers within the distributed system share no memory and communicate with each other by messages.

[0027] An aspect of this invention is to provide mechanisms for fault-tolerant systems based on replication, in which every operation is performed within two or more distinct replicas of an application process, object or component, typically located on different computers within a distributed system. In the event that one of the replicas is disabled by a fault, another replica can

continue to provide service.

[0028] Another aspect of this invention is to employ active replication, in which every operation is performed by two or more replicas of an application process, object or component, where each of those replicas has the same state, and where the replicas perform the same operations in the same order and thus continue to have the same state, i.e., they maintain strong replica consistency. If one of the replicas is disabled by a fault, another replica continues to provide service without any hiatus, as if no fault occurred. In an active replication strategy, all replicas are equal, and all replicas are treated alike.

[0029] There are many sources of nondeterminism in application programs. The mechanisms of this invention address nondeterminism caused by multithreading in replicated application programs that use the active replication strategy. They assume that the application program has been correctly coded so that each resource that is shared by two or more threads is protected by a mutex. They assume further that each thread has a unique thread identifier and that each mutex has a unique mutex identifier.

[0030] The mechanisms of this invention ensure that the threads in all of the replicas of an actively replicated multithreaded application program are granted their claims to mutexes, semaphores, and so forth in the same order, even though the threads in the various replicas might claim the mutexes, semaphores, and so forth in different orders. Thus, the mechanisms of this invention eliminate multithreading as a source of nondeterminism in an actively replicated multithreaded application program in order to maintain strong replica consistency.

[0031] The invention assumes, and exploits, the reliable ordered message delivery of a multicast group communication protocol to maintain strong replica consistency for an actively replicated multithreaded application program.

[0032] Another aspect of this invention is that, at each of the replicas, the mechanisms of the invention multicast special control messages, the PendingClaims messages, that contain mutex ordering information, namely,

the order in which the threads in the replicas claim mutexes. The order in which the multicast group communication protocol delivers the PendingClaims messages to the replicas determines the order in which the threads in the replicas are granted the mutexes.

[0033] In accordance with another aspect of the invention, to maintain application transparency for replicated multithreaded application programs based on the active replication strategy while maintaining strong replica consistency, the mechanisms of the invention employ the technique of library interpositioning to intercept the calls to functions of the operating system's thread library and to wrap the functions of the operating system's thread library.

[0034] One embodiment of the invention comprises a Consistent Multithreading (CMT) library that is interposed ahead of the operating system's thread library, such as the standard POSIX thread (PTHREAD) library. The CMT library contains wrapper functions for the functions of the operating system's thread library that claim and release mutexes, semaphores, condition variables, etc. The application program invokes the wrapper functions of the CMT library, instead of the corresponding functions of the operating system's thread library. The wrapper functions of the CMT library subsequently invoke the corresponding functions of the operating system's thread library. This allows the CMT library to modify the behavior of the replicated multithreaded application program, without modifying either the application program or the functions of the operating system's thread library.

[0035] When a replica of the replicated multithreaded program invokes a function to claim a mutex, the CMT claim() function is invoked. The CMT claim() function multicasts a PendingClaims message, to all of the replicas of a process, object or component using the reliable ordered multicast group communication protocol, and subsequently invokes the corresponding function of the operating system's thread library. The multicast protocol delivers messages reliably and in the same order to all of the replicas. If two different threads in a replica both issue a claim for a mutex, the message containing the claim, that the multicast protocol orders and delivers first,

determines which thread is granted the mutex first. Consequently, the mechanisms of the invention grant the mutexes in the same order to the threads in all of the replicas. In effect, they sanitize the non-deterministic behavior of the threads in the replicas when they invoke the claim() function of the operating system's thread library and, similarly, for the release() function and for semaphores, control variables, and so forth.

[0036] For application programs that run on an operating system that provides Dynamically Linked Libraries (DLL) (e.g., Solaris, Linux, Windows), a command is issued to the DLL mechanisms that causes the DLL mechanisms to interpose the Consistent Multithreading (CMT) library, containing the wrapper functions, ahead of the operating system's thread library. This interpositioning causes the application program to invoke the functions of the CMT library, rather than the corresponding functions of the operating system's thread library directly. In this case, the mechanisms involve no modification or recompilation of the application program and, thus, are transparent to the application program.

[0037] If, on the other hand, the operating system does not provide Dynamically Linked Libraries (e.g., VxWorks), it is necessary to insert a statement into the makefile for the application program that directs the linker to include the CMT library ahead of the operating system's thread library. Thus, in this case, the application program is not modified but the makefile is modified.

[0038] The mechanisms of this invention allow concurrency of threads that do not simultaneously acquire the same mutex, while maintaining strong replica consistency. Thus, they allow the maximum degree of concurrency possible, while maintaining strong replica consistency.

[0039] The mechanisms of this invention sanitize multithreaded application programs in that they mask multithreading as a source of non-determinism so that strong replica consistency is maintained.

[0040] The invention may be described in a number of alternative ways, the following being provided by way of example. The invention teaches a system for executing threads in replicas of application programming that are

replicated according to the active replication strategy. The system comprises: (1) means for communicating to multiple replicas the claims of shared resources by threads in a replica; and (2) means for ordering shared resource claims to be granted to threads in multiple replicas corresponding to the order in which the shared resource claims were communicated, ordered and delivered through the means for communicating the order of claiming. The shared resources may comprise shared data, code or input/output locations accessible to threads of the replica. By way of example, access to the shared resources may be controlled by mutexes, or similar access control mechanisms.

[0041] The means for communicating is configured to multicast messages containing information about which shared resource is being claimed, which thread is claiming the given shared resource, and which shared resource claim request, such as a shared resource claim number, of the thread is being made.

[0042] The means for ordering is configured to prevent threads from being granted a shared resource until claim information has been communicated to the replicas by at least one of the replicas. The means for ordering shared resource claims is configured to maintain an identical claim granting order across the replicas, such as according to a routine for selectively granting a resource request with a particular resource claim number, based on the order in which that resource request was communicated, ordered and delivered to the replicas. The means for ordering may comprise at least one allocation routine configured for granting access to shared resources to threads in the replica, in response to information delivered about the order in which threads in the replica and other replicas claim access to shared resources.

[0043] A method according to the present invention generally describes executing actively replicated multithreaded application programs containing threads that access shared resources, comprising: (1) issuing a claim request for a shared resource by a thread; (2) multicasting a message to communicate the shared resource claim to all of the replicas of the application program; (3) ordering shared resource claim requests; and (4) granting the shared resource

to a given thread in response to the order in which multicast messages are communicated, ordered and delivered, and to the availability of the shared resource.

[0044] The invention may be practiced in a number of alternative ways including embodied in an apparatus, system, method, library, software, media containing software programming, and so forth without departing from the teachings of the present invention.

[0045] It can be seen, therefore, that the present invention differs from prior strategies in several ways. For example, it will be appreciated that the present invention applies to active replication in which all replicas are equal, whereas strategies such as detailed in U.S. Patents 4,718,002, 5,621,885, 5,802,265 and 5,968,185 require casting replicas into Primary or Backup roles. Furthermore, the library interpositioning and wrapping approach of the current invention is distinctly different than utilizing object code editing to modify the program code as is done in U.S. Patents 5,802,265 and 5,968,185 and the TFT system where there are no mechanisms for active replication. Note also that the present invention does not require the use of hardware- or software-based instruction counters and thus differs from the strategy such as proposed by J. H Sly and E. N. Elnozahy described previously. Moreover, the current invention does not impose a single logical thread of control on the replicas in order to maintain strong replica consistency, such as Transactional Drago and other systems described above.

[0046] Further aspects of the invention will be brought out in the following portions of this document, wherein the detailed description is for the purpose of fully disclosing preferred embodiments of the invention without placing limitations thereon.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWING(S)

[0047] The invention will be more fully understood by reference to the following drawings which are for illustrative purposes only.

[0048] FIG. 1 is a diagram showing two replicas each executing multiple threads that share data, which are protected by mutexes, and where a Consistent Multithreading (CMT) library and a reliable ordered multicast group

communication protocol are shown according to an embodiment of the invention.

[0049] FIG. 2 is a process flow diagram showing two replicas each executing multiple threads, with a history of the order in which mutexes are claimed, granted and released, and in which PendingClaims messages are multicast according to an embodiment of the invention.

[0050] FIG. 3 is a flow chart that shows a thread of a replica invoking the CMT claim() function for a mutex and the steps taken by that function when it is invoked according to an embodiment of the invention.

[0051] FIG. 4 is a flow chart that shows a thread of a replica invoking the CMT release() function for a mutex and the steps taken by that function when it is invoked according to an embodiment of the invention.

[0052] FIG. 5 is a flow chart that shows a replica receiving a PendingClaims message and the steps taken by the CMT message handler when it receives that message according to an embodiment of the invention.

[0053] FIG. 6 is a flow chart that shows a thread of a replica that is awakened while waiting for a mutex and the steps taken by the CMT claim() function according to an embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

Introduction

[0054] Referring more specifically to the drawings, for illustrative purposes the present invention is embodied in the processes and systems shown in FIG. 1 through FIG. 6. Those skilled in the art will appreciate that the invention may be implemented in other ways than those shown and described. In the algorithms, diagrams and descriptions, the replicas of the replicated application program may be processes, objects, components or other such entities. The term *mutex* refers to a binary semaphore. However, the mechanisms of the invention apply equally well to counting semaphores, condition variables, and other shared resource access control mechanisms. The present invention uses known process management functions such as claim(), release(), suspend() and signal() functions. In the methods, diagrams and descriptions below, the code that wraps the functions of the operating

system's thread library is referred to as the *Consistent Multithreading (CMT) library.*

Library Interpositioning

[0055] In general terms, the mechanisms of this invention employ a technique of library interpositioning to intercept the calls to functions of the operating system's thread library. For example, for the standard POSIX Thread (PTHREAD) library, the mechanisms intercept the calls to the functions of the PTHREAD library.

[0056] In the preferred embodiment of the invention, the Consistent Multithreading (CMT) library contains wrappers for the functions of the operating system's thread library, such as the pthread_mutex_lock() and pthread_mutex_unlock() functions of the PTHREAD library. Because the mechanisms of this invention apply to threading libraries other than the PTHREAD library, we refer to these methods, and the corresponding wrapper functions, more generally as claim() and release(), respectively.

[0057] When a thread of a replica claims a mutex, the thread actually invokes the claim() function of the CMT library, instead of the claim() function of the operating system's thread library. The CMT claim() function subsequently invokes the claim() function of the operating system's thread library. However, before doing so, the CMT claim() function takes appropriate steps, described below, to ensure that the threads in the different replicas claim the mutexes in the same order.

[0058] The CMT mechanisms multicast a PendingClaims message containing a sequence of (T, M, N) identifier triples (the mutex ordering information), where thread T of a replica has claimed mutex M and this claim is T's Nth claim of any mutex. Thread T of a replica is not granted its claim of mutex M until the reliable ordered multicast protocol orders and delivers the message containing the mutex ordering information (T, M, N) to the replicas. The reliable ordered multicast protocol guarantees that the message delivery order of the PendingClaims messages is the same at all of the replicas.

[0059] For each mutex M, the CMT mechanisms maintain a Boolean variable, M.available, which indicates whether or not mutex M is available. For each

thread T, the CMT mechanisms maintain a Boolean variable, T.suspended, which indicates whether or not thread T is suspended. For each thread T, the CMT mechanisms maintain an integer variable T.granted which is set to N, where the claim being granted is T's Nth claim of any mutex.

[0060] The CMT mechanisms at each replica maintain a queue, the *PendingClaims queue*, of outstanding (T, M, N) triples, where thread T has claimed mutex M and this claim is T's Nth claim of any mutex. The PendingClaims queue spans different threads, different mutexes and different claims of the different mutexes by the different threads. As the threads claim the mutexes, the CMT mechanisms intercept the calls to the operating system's claim() function and append the (T, M, N) triples to the PendingClaims queue.

[0061] The CMT mechanisms at each replica multicast a special control message, the *PendingClaims message*, that contains information about the order in which the threads in that replica claim mutexes. The order in which messages are delivered determines the order in which the threads are granted mutexes. For example, if a PendingClaims message from one replica contains (T, M, N) and a PendingClaims message from another replica contains (T', M, N'), then the first such message ordered and delivered determines whether thread T or thread T' is granted the mutex M first.

[0062] For each mutex M, the CMT mechanisms at each replica maintain a queue, the *M.orderedClaims queue*, of claims that have been ordered and delivered. The entries of the M.orderedClaims queue occur in the order in which the claims of the different threads for the mutex M are granted. The CMT mechanisms grant, to the threads, the mutexes that they claim based on the order in which those claims occur in the M.orderedClaims queue, rather than on the order in which the threads are suspended, or on the order determined by the operating system scheduler on the local computer.

Two Multithreaded Replicas with the Interposed CMT Library

[0063] Referring now to FIG. 1, two multithreaded replicas, R1 and R2, are shown 10, each executing in its own process 12, 14 respectively. A reliable ordered multicast group communication protocol 36 conveys messages to

replicas R1 and R2 and delivers the messages reliably and in the same order (linear sequence) to both of the replicas.

[0064] In the example shown, there are two sets of threads 16, 18 associated with replica R1 and replica R2, respectively. In replica R1, the thread set 16 comprises four threads T1-T4 that are designated as 20-26, respectively. Thread T1 20 and thread T2 22 access the shared resource 38 comprising shared data D1 44 using mutex M1 46. Thread T3 24 and thread T4 26 access the shared resource 40 comprising shared data D2 48 using mutex M2 50. The CMT library 42 is interposed between replica R1 and the operating system's thread library, and is dynamically or statically linked into replica R1.

[0065] Similarly, in replica R2, the thread set 18 comprises four threads T1-T4 that are designated as 28-34, respectively. Thread T1 28 and thread T2 30 access the shared resource 52 comprising the shared data D1 58 using mutex M1 60. Thread T3 32 and thread T4 34 access the shared resource 54 comprising shared data D2 62 using mutex M2 64. The CMT library 56 is interposed between replica R2 and the operating system's thread library, and is dynamically or statically linked into replica R2.

[0066] In replica R1, because thread T1 20 and thread T2 22 can each read and write the shared data D1 44, their access to that data is protected by mutex M1 46. Similarly, thread T3 24 and thread T4 26 can each read and write the shared data D2 48, to which access is protected by mutex M2 50. However, threads T1 20 and T2 22 share no data with threads T3 24 and T4 26; thus, threads T1 20 and T2 22 can execute concurrently with threads T3 24 and T4 24 without the need for a mutex. The same is true for the corresponding threads in replica R2.

Example of the CMT Mechanisms

[0067] FIG. 2 represents a scenario that involves two replicas R1 70 and R2 72 which are each executing multiple threads T1 74, T2 76, T3 78 and T1 80, T2 82, T3 84, respectively. In typical scenarios, the replicas comprise two or more replicas of a process, object or component hosted on two or more computers within a distributed computing environment. The scenario shows a history of the order in which mutexes are claimed, granted and released, using

the mechanisms of the CMT library, according to an aspect of the present invention. The mutexes are controlled by the CMT library functions, instead of only by the corresponding functions of the operating system's thread library, in order to maintain strong replica consistency.

[0068] The CMT mechanisms communicate pending claims for mutexes to multiple replicas, such as according to the use of multicast messages. The means for communicating the mutex operations are generally provided by way of PendingClaims messages that are multicast to the replicas, according to an aspect of the present invention.

[0069] Each thread is shown as a sequence of instruction executions, which include access to shared data. The M.orderedClaims queue for mutex M, associated with the shared data, is shown in the figure at specific times, such as when a claim for mutex M is ordered, delivered and placed in the M.orderedClaims queue, or when a claim for mutex M is granted and removed from the queue.

[0070] It should be appreciated that the figure represents a simple case for clarity, while numerous claims for mutex M may be pending for a number of threads, wherein the M.orderedClaims queue could contain many entries. Additionally, the reliable ordered multicast messages are shown by way of example as a small sequence of multicast messages, although many messages may additionally be multicast.

[0071] Referring to the figure, replica R1 70 is shown by way of example executing threads T1 74, T2 76 and T3 78, and replica R2 72 is shown by way of example executing threads T1 80, T2 82 and T3 84. Threads T1 74 and T2 76 of replica R1 access shared data via mutex M 86. Thread T3 78 does not share data with thread T1 74 or thread T2 76, and consequently it executes concurrently with these threads without the need for a mutex. Similarly, threads T1 80 and T2 82 in replica R2 access the shared data via mutex M 88. Thread T3 84 does not share data with thread T1 80 or thread T2 82 and as a result executes concurrently with these threads without the need for a mutex.

[0072] When thread T1 74 in replica R1 70 invokes the CMT claim() function

to claim 90 mutex M 86 the CMT claim() function checks whether its M.orderedClaims queue 92 already contains a claim 94 for mutex M by thread T1. In this case, the M.orderedClaims queue 92 does not currently contain such a claim, wherein the CMT claim() function thereby multicasts 96 a PendingClaims message containing the claim (T1, M, 8) 98 and suspends thread T1 74.

[0073] Similarly, when thread T2 76 in replica R1 70 invokes the CMT claim() function to claim 102 mutex M 86, the CMT claim() function checks whether the M.orderedClaims queue 92 already contains a claim 102 for mutex M by thread T2 . As the M.orderedClaims queue 92 does not currently contain such a claim, the CMT claim() function multicasts 96 a PendingClaims message containing the claim (T2, M, 5) 104 and suspends thread T2 76.

[0074] Concurrently, in replica R2 72, when thread T2 82 invokes the CMT claim() function to claim 106 mutex M, the CMT claim() function checks whether its M.orderedClaims queue 108 already contains the claim (T2, 5) 110. As the M.orderedClaims queue 108 does not currently contain such a claim, the CMT claim() function multicasts a PendingClaims message containing the claim (T2, M, 5) 112, and suspends thread T2 82.

[0075] When replica R1 70 receives a PendingClaims message that contains a claim (T1, M, 8), the CMT message handler extracts the claim (T1, 8) and checks whether its M.orderedClaims queue 92 already contains the claim (T1, 8). As its M.orderedClaims queue does not contain the claim (T1, 8), the CMT message handler places that claim in the M.orderedClaims queue which then contains only the claim (T1, 8) 114. Because thread T1 74 is currently suspended waiting for mutex M 86 as a result of its previous claim (T1, 8) 90, and because no thread currently holds mutex M, the CMT mechanisms awaken thread T1 74 in replica R1 70.

[0076] When thread T1 74 in replica R1 70 awakens, the CMT claim() function checks whether (T1, 8) is the first entry in its M.orderedClaims queue 92. As (T1, 8) is the first entry 114 in the M.orderedClaims queue, the CMT claim() function removes claim (T1, 8) from that queue, which then becomes empty 116, and grants 118 mutex M to thread T1 74, thereby allowing thread T1 to

enter the critical section 120 and access the shared data.

[0077] In replica R2 72, it should be noted that thread T2 82, instead of thread T1 80, claims mutex M first. That order is inconsistent with the order in replica R1, where thread T1 claimed, and was granted, the mutex first. The CMT library, as an embodiment of this invention, enforces consistent ordering of mutex claims and claim releases which, in this scenario, means that thread T2 in replica R2 will not be granted its claim to mutex M until thread T1 in replica R2 is granted its claim to mutex M.

[0078] When replica R1 70 receives the PendingClaims message containing pending claim (T2, M, 5), which replica R2 72 generated as a consequence of thread T2 claiming 104 mutex M, and which the multicast protocol ordered and delivered to replica R1, the CMT message handler at replica R1 checks whether its M.orderedClaims queue 92 already contains the claim (T2, 5). Because the M.orderedClaims queue does not contain that claim, the CMT message handler at replica R1 appends the claim (T2, 5) to the empty M.orderedClaims queue 116, which then contains one entry 122. However, thread T1 74 of replica R1 70 already holds mutex M 86; therefore, the CMT mechanisms do not awaken thread T2 76 of replica R1.

[0079] When replica R1 70 receives the PendingClaim message containing the claim (T2, M, 5) 104, based on replica R1's claim (T2, 5) 102, the CMT message handler checks whether the M.orderedClaims queue 92 already contains claim (T2, 5). Because claim (T2, 5) is already contained in the M.orderedClaims queue 122, the CMT message handler discards the PendingClaims message containing claim (T2, 5), because it is a duplicate, wherein the M.orderedClaims queue 124 is identical to the prior M.orderedClaims queue 122.

[0080] When replica R2 72 receives a PendingClaims message containing the claim (T1, M, 8) 98 from replica R1 70, the CMT message handler at replica R2 checks whether the claim (T1, 8) is already in its M.orderedClaims queue 108. As claim (T1, 8) is not in its M.orderedClaims queue, the CMT message handler appends claim (T1, 8) to that queue, which now contains one entry (T1, 8) 126. As the thread T1 80 in replica R2 72 is not suspended, the CMT

mechanisms do not awaken thread T1.

[0081] When replica R2 72 receives the PendingClaims message containing claim (T2, M, 5) 112 that replica R2 multicast and that the multicast protocol ordered and delivered, the CMT message handler at replica R2 checks whether the claim (T2, 5) is already in its M.orderedClaims queue 108. Because the claim (T2, 5) is not in its M.orderedClaims queue, the CMT message handler at replica R2 appends (T2, 5) to its M.orderedClaims queue, which then contains two entries 128. As the claim (T2, 5) is not the first entry in the M.orderedClaims queue at replica R2 72, the CMT mechanisms at replica R2 do not awaken thread T2 82.

[0082] When replica R2 72 receives the PendingClaims message containing the claim (T2, M, 5) 104 that replica R1 70 multicast and that the multicast protocol ordered and delivered, the CMT message handler at replica R2 checks whether the claim (T2, 5) is already in its M.orderedClaims queue 108. Because (T2, 5) is already in its M.orderedClaims queue, the CMT message handler at replica R2 discards that claim as a duplicate, wherein its M.orderedClaims queue 130 is the same as its M.orderedClaims queue at 128.

[0083] When thread T1 80 in replica R2 72 invokes the CMT claim() function to claim 132 mutex M, the CMT claim() function checks whether the claim (T1, 8) is already in its M.orderedClaims queue. Because the claim (T1, 8) is already in its M.orderedClaim queue 130, the CMT claim() function discards the duplicate claim and does not multicast it. Because no thread in replica R2 72 is currently holding mutex M 88, the CMT claim() function removes the claim (T1, 8) from the M.orderedClaims queue, which then contains one entry (T2, 5) 134, and grants 136 mutex M to thread T1. It then returns, allowing thread T1 80 to execute the critical section 138 of code in which it accesses the shared data that mutex M protects.

[0084] In replica R1 70, when thread T1 74 has finished accessing the shared data protected by mutex M 86 and invokes the release function, the CMT release() function 140 is invoked. The CMT release() function marks mutex M as available, and checks whether another thread is currently waiting for that

mutex. Because the first entry in its M.orderedClaims queue currently contains the claim (T2, 5) 145, the CMT release() function determines that thread T2 76 is waiting for mutex M 86. The CMT release() function then awakens thread T2 76 and returns execution to thread T1 74 allowing it to proceed.

[0085] When thread T2 76 in replica R1 70 awakens, the CMT claim() function removes the claim (T2, 5) from its M.orderedClaims queue, which then becomes empty 144 and grants mutex M to thread T2 146 and returns, thereby allowing thread T2 76 to proceed.

[0086] In replica R2 72, when thread T1 80 has finished accessing the shared data protected by mutex M and invokes the release function to release mutex M, the CMT release() function 148 is invoked. The CMT release() function marks mutex M 88 as available, and checks whether another thread is waiting for that mutex. As the M.orderedClaims queue 108 contains the claim (T2, 5) 150, which indicates that thread T2 82 is waiting for mutex M, the CMT release() function then awakens thread T2. The CMT release() function then returns execution to thread T1 80, allowing T1 to proceed.

[0087] When thread T2 82 in replica R2 72 awakens, the CMT claim() function checks whether the claim (T2, 5) is the first entry 150 in its M.orderedClaims queue. As (T2, 5) is the first entry in its M.orderedClaims queue, the CMT claim() function removes the claim (T2, 5) from that queue, which then becomes empty 152. The CMT claim() function then grants 154 mutex M to thread T2 82 in replica R2 72 and returns, thereby allowing thread T2 to proceed.

Replica Thread Invokes CMT Claim() Function to Claim a Mutex

[0088] At a replica, when thread T invokes the claim function to claim mutex M, it actually invokes the CMT claim() function. The CMT claim() function then executes the following steps:

```
determine (T, M, N)
if (T, N) is the first entry in the M.orderedClaims queue
        remove (T, N) from the M.orderedClaims queue
        set T.granted to N
```

```
            set M.available to false
            grant M to T
        else
            if (T, N) does not occur anywhere in the M.orderedClaims queue
                append (T, M, N) to the PendingClaims queue of claims
                to be multicast
        set T.suspended to true
        suspend T
```

Thus, referring to FIG. 3, at a replica, when thread T invokes the CMT claim() function to claim mutex M and this claim is T's Nth claim of any mutex at block 210, the CMT claim() function first determines the triple (T, M, N) at block 212.

[0089] It then checks whether the pair (T, N) is the first entry in the M.orderedClaims queue at block 214. If the pair (T, N) is the first entry in that queue, the CMT claim() function then removes (T, N) from that queue at block 216, sets T.granted to N at block 218, sets M.available to false at block 220, grants M to T at block 222 and then returns at block 224.

[0090] If at block 214 the pair (T, N) is not the first entry in the M.orderedClaims queue, the CMT claim() function checks whether the pair (T, N) occurs anywhere in the M.orderedClaims queue at block 226. If the pair (T, N) does not occur in that queue, the CMT claim() function appends (T, M, N) to the PendingClaims queue of claims to be multicast at block 228. In either case, it then sets T.suspended to true at block 230 and suspends thread T at block 232.

[0091] Periodically, or immediately when the CMT mechanisms add an entry to the PendingClaims queue, the CMT mechanisms multicast a PendingClaims message containing the entries of the PendingClaims queue, as shown at 96 in FIG. 2.

Replica Multicasts a PendingClaims Message

[0092] Periodically, or immediately when an entry is added to the PendingClaims queue, the CMT mechanisms at each replica multicast a PendingClaims message, as shown at 98, 112 and 104 in FIG. 2.

Replica Thread Invokes CMT Release() Function to Release a Mutex

[0093] When a thread T in a replica invokes the release function to release mutex M, it actually invokes the CMT release() function. The CMT release() function executes the following steps:

```
set M.available to true
if the M.orderedClaims queue is not empty
    determine the first entry (T', N') in the M.orderedClaims queue
    if T'.suspended
        signal T' to awaken it
```

Thus, referring to FIG. 4, when a thread T invokes the CMT release() function to release mutex M at block 250, first the CMT release() function sets M.available to true at block 252 and then it checks whether the M.orderedClaims queue is empty. If the M.orderedClaims queue is not empty, it determines the first entry (T', N') in the M.orderedClaims queue at block 256 and then checks whether T'.suspended is true at block 258. If T' is suspended, the CMT release() function signals T' to awaken it at block 260 and then returns at block 262. Otherwise, if T' is not suspended or if the M.orderedClaims queue is empty, the CMT release() function simply returns at block 262.

Replica Thread Is Awakened

[0094] When a thread T is awakened while waiting for mutex M as its Nth claim of any mutex, the CMT claim() function executes the following steps:

```
if (T, N) is the first entry in the M.orderedClaims queue
    remove (T, N) from the M.orderedClaims queue
    set T.granted to N
    set M.available to false
    grant M to T
else
    suspend T
```

Thus, referring to FIG. 5, when a thread T is awakened while waiting for mutex M as its Nth claim of any mutex at block 270, the CMT claim() function checks whether (T, N) is the first entry in the M.orderedClaims queue at block 272. If

so, the CMT claim() function removes (T, N) from the M.orderedClaims queue at block 274, sets T.granted to N at block 276 which records T's Nth claim as having been granted, sets M.available to false at block 278 and grants M to T at block 280, and then resumes T at block 282. If (T, N) is not the first entry in the M.orderedClaims queue, the CMT claim() function suspends T at block 284.

Replica Receives a PendingClaims Message

[0095] When the CMT message handler at a replica receives a PendingClaims message containing the mutex ordering information, it extracts that information. For each entry (T, M, N) extracted from the PendingClaims message, the CMT message handler performs the following steps:

```
if (T, N) is not in the M.orderedClaims queue and if N > T.granted
(i.e., T's Nth claim of a mutex has not been granted)
    append (T, N) to the M.orderedClaims queue
    if M.available and T.suspended
        signal T to awaken it
```

Thus, referring to FIG. 6, when a replica receives a PendingClaims message that contains the mutex ordering information at block 290, it extracts that information. For each entry (T, M, N) extracted from the PendingClaims message, the replica checks whether (T, N) is anywhere in its M.orderedClaims queue at block 292. If (T, N) is not in the M.orderedClaims queue and if N > T.granted (i.e., T's Nth claim has not already been granted) at block 294, the CMT message handler appends (T, N) to the M.orderedClaims queue at block 296. It then checks whether M is available at block 298 and T is suspended at block 300. If so, it signals T to awaken it at block 302 and then terminates at block 304. In all other cases, it simply terminates at block 304.

[0096] As can be seen, therefore, the present invention comprises a method and system for transparent consistent active replication of multithreaded application programs wherein multithreading is sanitized or masked in a transparent manner by intercepting calls to functions, such as functions of a Consistent Multithreading library (CMT), that claim and release mutual

exclusion constructs, or similar access control structures. When a thread of a replica of the application program claims or releases a mutual exclusion construct that protects a shared resource, it actually invokes the claim function of the CMT library that wraps the corresponding function of the operating system's thread library. The claim function of the CMT library multicasts a message, containing the claim for the mutual exclusion construct, to all replicas. The order in which the claim for the mutual exclusion construct is granted to a thread in a replica is determined competitively by the order in which the multicast messages, containing claim information for the mutual exclusion construct, are communicated and delivered to that replica. Because the replicas receive the same messages in the same source order, the corresponding threads in the different replicas are granted their corresponding claims to the corresponding mutual exclusion constructs in the same order, maintaining strong replica consistency.

[0097] Although the description above contains many details, these should not be construed as limiting the scope of the invention but as merely providing illustrations of some of the presently preferred embodiments of this invention. Therefore, it will be appreciated that the scope of the present invention fully encompasses other embodiments which may become obvious to those skilled in the art, and that the scope of the present invention is accordingly to be limited by nothing other than the appended claims, in which reference to an element in the singular is not intended to mean "one and only one" unless explicitly so stated, but rather "one or more." All structural and functional equivalents to the elements of the above-described preferred embodiment that are known to those of ordinary skill in the art are expressly incorporated herein by reference and are intended to be encompassed by the present claims. Moreover, it is not necessary for a device or method to address each and every problem sought to be solved by the present invention, for it to be encompassed by the present claims. Furthermore, no element, component, or method step in the present disclosure is intended to be dedicated to the public regardless of whether the element, component, or method step is explicitly recited in the claims. No claim element herein is to be construed

under the provisions of 35 U.S.C. 112, sixth paragraph, unless the element is expressly recited using the phrase "means for."

CLAIMS

What is claimed is:

1. A method for replicating a multithreaded application program with an active replication strategy, wherein said application program executes under the control of an operating system having a thread library, the method comprising:

at each replica, multicasting control messages that contain mutex ordering information indicating the order in which threads in the replicas claim mutexes; and

delivering said control messages using a multicast group communication protocol that delivers the messages in an order that determines the order in which the operating system's thread library grants said claims of mutexes to the threads in the replicas.

2. A method as recited in claim 1, further comprising:

employing thread library interpositioning to intercept calls to functions of the operating system's thread library;

wherein said interpositioning is provided by a Consistent Multithreading (CMT) library interposed ahead of the operating system's thread library, so that, when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said library interpositioning to a function of the CMT library which subsequently invokes the corresponding function of the operating system's thread library.

3. A method for replicating a multithreaded application program with an active replication strategy, wherein said multithreaded application program executes under the control of an operating system having a thread library, the method comprising:

employing thread library interpositioning to intercept calls to functions of the operating system's thread library;

wherein said interpositioning is provided by a Consistent Multithreading (CMT) library interposed ahead of the operating system's thread library, so that when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said library interpositioning to

a function of the CMT library which subsequently invokes the corresponding function of the operating system's thread library.

4. A method as recited in claim 3, further comprising:

at each replica, multicasting control messages that contain mutex ordering information specifying the order in which threads in the replicas claim mutexes; and

delivering said control messages using a multicast group communication protocol that delivers the messages in an order that determines the order in which the operating system's thread library grants said claims to the threads in the replicas.

5. A method for replicating a multithreaded application program with an active replication strategy, wherein said application program executes under the control of an operating system having a thread library, the method comprising:

at each replica, multicasting control messages that contain mutex ordering information specifying the order in which threads in the replica claim mutexes;

delivering said control messages using a multicast group communication protocol that delivers the messages in an order that determines the order in which the operating system's thread library grants said claims to the threads in the replicas; and

employing thread library interpositioning to intercept calls to functions of the operating system's thread library;

wherein said interpositioning is provided by a Consistent Multithreading (CMT) library that is interposed ahead of the operating system's thread library, so that when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said library interpositioning to a function of the CMT library which subsequently invokes the function of said operating system's thread library.

6. A method for replicating a multithreaded application program with an active replication strategy, wherein said application program executes under the control of an operating system having a thread library, the method comprising:

providing a Consistent Multithreading (CMT) library that is interposed ahead of

the operating system's thread library.

7. A method as recited in claim 6, wherein said CMT library contains wrapper functions for functions of the operating system's thread library.

8. A method as recited in claim 7, wherein in response to a replica of said replicated multithreaded application program invoking a function to claim or release a mutex, said invocation is diverted by library interpositioning to a function of the CMT library which subsequently invokes a function of the operating system's thread library.

9. A method as recited in claim 8, wherein said CMT library function multicasts a message to all of the replicas of a process, object or component and subsequently invokes the corresponding function of the operating system's thread library.

10. A method as recited in claim 9, wherein said message is multicast using a reliable ordered multicast group communication protocol.

11. A method as recited in claim 10:

wherein said multicast protocol delivers messages reliably and in the same order to all of the replicas;

wherein the mutexes are granted in the same order to the threads in all of the replicas.

12. A method as recited in claim 9, wherein if two different threads both request a particular claim for a mutex, the message containing that claim that is ordered and delivered by the multicast protocol first, determines the order in which the operating system's thread library grants said claims to the threads in the replicas first.

13. A method as recited in claim 6:

wherein if the application program runs on an operating system that supports Dynamically Linked Libraries (DLL), the DLL mechanisms are used to interpose the CMT library ahead of the operating system's thread library;

wherein, when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, the invocation is diverted by said DLL mechanisms to a function of the CMT library which subsequently invokes a function of the operating system's thread library.

14. A method as recited in claim 6, further comprising:

adding a statement to the makefile for said application program that causes the linker to interpose the CMT library ahead of the operating system's thread library if the operating system does not provide Dynamically Linked Libraries;

wherein, when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by library interpositioning to a function of the CMT library which subsequently invokes a function of the operating system's thread library.

15. A method for achieving strong replica consistency for replicated multithreaded application programs that use an active replication strategy, comprising:

sanitizing replicated multithreaded application programs by masking multithreading as a source of non-determinism.

16. A method for replicating a multithreaded application program using an active replication strategy, wherein said application program executes under the control of an operating system having a thread library, the method comprising:

providing a Consistent Multithreading (CMT) library that is interposed ahead of the operating system's thread library;

wherein said CMT library contains wrapper functions for functions of the operating system's thread library;

wherein, when a replica of said replicated multithreaded application program

invokes a function to claim or release a mutex, said invocation is diverted by said interpositioning to a wrapper function of the CMT library which subsequently invokes the function of the operating system's thread library.

17. A method as recited in claim 16, wherein when a replica invokes a function of the CMT library to claim a mutex, the CMT library function multicasts a message containing that claim to all of the replicas of the process, object or component and subsequently invokes the corresponding function of the operating system's thread library.

18. A method as recited in claim 17:

wherein said message is multicast using a reliable ordered multicast group communication protocol;

wherein said multicast protocol delivers messages reliably and in the same order to all of the replicas of the process, object or component.

19. A method as recited in claim 18, wherein the mutexes are granted in the same order to the threads in all of the replicas of the process, object or component.

20. A method as recited in claim 19, wherein if two different threads both issue a particular claim for a mutex, the message containing said claim that the multicast protocol orders and delivers first determines which claim the operating system's thread library grants first.

21. A method as recited in claim 16:

wherein if the application program runs on an operating system that provides Dynamically Linked Libraries (DLL), the DLL mechanisms are used to interpose the CMT library ahead of the operating system's thread library;

wherein, when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said interpositioning to a function of the CMT library which subsequently invokes a

function of the operating system's thread library.

22. A method as recited in claim 16, further comprising:

adding a statement to the makefile for said application program that causes the linker to interpose the CMT library ahead of the operating system's thread library if the operating system does not provide Dynamically Linked Libraries;

wherein, when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said interpositioning to a function of the CMT library which subsequently invokes a function of the operating system's thread library.

23. A method for replicating a multithreaded application program with an active replication strategy, wherein said application program executes under the control of an operating system, said method comprising:

allowing threads to communicate with each other by multicasting messages; and

allowing threads to use a shared resource.

24. A method as recited in claim 23, wherein said shared resource comprises shared data.

25. A method as recited in claim 23, further comprising:

using a mutex to protect said shared resource accessed by threads in the replicas of said replicated multithreaded application program;

wherein said threads in all replicas of the multithreaded application program access the shared resource in the same order.

26. A method as recited in claim 25, further comprising:

intercepting calls to the operating system's thread library at each replica; and

multicasting, to the replicas, ordering information regarding the order in which threads in the replicas claim mutexes.

27. A method as recited in claim 26, wherein multicast messages that contain ordering information regarding the order in which threads claim mutexes are delivered reliably and in the same order to all of the replicas of a process, object or component.

28. A method as recited in claim 27, wherein if multicast messages from two different replicas contain ordering information such that each of two different threads in said replicas claim the same mutex, then the message that is ordered and delivered first determines which thread in the replicas is granted its claim to the mutex first.

29. A method as recited in claim 28, further comprising:

maintaining strong replica consistency and application transparency by interpositioning a multithreading library ahead of the operating system's thread library and intercepting calls to functions of said operating system's thread library, so as to render the application program virtually deterministic.

30. A method as recited in claim 29:

wherein functions of the operating system's thread library are wrapped by functions of the multithreading library;

wherein the wrapper functions invoke the corresponding functions of the operating system's thread library.

31. A method as recited in claim 30, wherein said wrapping is performed by functions of a multithreading library that is dynamically linked to said application program.

32. A method as recited in claim 30, further comprising:

providing a mechanism at a replica that can decide not to multicast information, related to a thread of that replica claiming a mutex, if the replica has already received a message that contains information related to the same thread in another replica issuing the same claim for that mutex.

33. A method as recited in claim 32, wherein if a replica receives a message containing information related to a thread in that replica claiming a mutex, and if the replica has already received a message that contains information related to the corresponding thread in another replica issuing the same claim for the mutex, the replica ignores the information contained in the later received message.

34. A method as recited in claim 33, further comprising:

allowing concurrent processing of threads that do not attempt to use the same shared resource simultaneously; and

allowing concurrent processing of threads that claim different mutexes;

wherein strong replica consistency is maintained.

35. A software mechanism for replicating a multithreaded application program using an active replication strategy, wherein said application program executes under the control of an operating system having a thread library, the mechanism comprising:

control program code;

said control program code configured, at each replica, to multicast control messages that contain mutex ordering information indicating the order in which threads in the replica claim mutexes;

said control program code delivering said control messages using a multicast group communication protocol that delivers the messages in an order that determines which claim the operating system's thread library grants first.

36. A software mechanism as recited in claim 35:

wherein a multithreading library containing said control program code is interpositioned ahead of the operating system thread's library; and

wherein when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said interpositioning to a function of the multithreading library which subsequently invokes a function of the operating system's thread library.

37. A software mechanism for replicating a multithreaded application program with an active replication strategy, wherein said application program executes under the control of an operating system having a thread library, the mechanism comprising:

a consistent multithreading thread library interpositioned ahead of the operating system's thread library so that when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said interpositioning to a function of a Consistent Multithreading (CMT) library which subsequently invokes a function of the operating system's thread library.

38. A software mechanism as recited in claim 37, further comprising:

control program code associated with said consistent multithreading thread library;

said control program code configured, at each replica, to multicast control messages that contain mutex ordering information that indicates the order in which threads in the replica claim mutexes;

said control program code configured to deliver said control messages using a multicast group communication protocol that orders and delivers messages in an order that determines which claim the operating system's thread library grants first.

39. A software mechanism for replicating a multithreaded application program with an active replication strategy, wherein said application program executes under the control of an operating system having a thread library, the mechanism comprising:

control program code;

said control program code configured, at each replica, to multicast control messages that contain mutex ordering information that indicates the order in which threads in the replica claim mutexes;

said control program code configured to deliver said control messages using a multicast group communication protocol that orders and delivers the messages in an order that determines which claim the operating system's thread library grants first;

and

a consistent multithreading library interpositioned ahead of the operating system's thread library to intercept calls to functions of the operating system's thread library.

40. A software mechanism for replicating a multithreaded application program with an active replication strategy, wherein said application program executes under the control of an operating system having a thread library, the mechanism comprising:

a Consistent Multithreading (CMT) library that is interposed ahead of the operating system's thread library;

wherein said Consistent Multithreading (CMT) library contains wrapper functions for functions of the operating system's thread library.

41. A software mechanism as recited in claim 40, wherein, when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said interpositioning to a function of the CMT library which subsequently invokes a function of the operating system's thread library.

42. A software mechanism as recited in claim 41, wherein when a replica of a replicated multithreaded application program claims a mutex, the CMT library function, while processing said claim, multicasts a message to all of the replicas of a process, object or component and subsequently invokes the corresponding function of the operating system's thread library.

43. A software mechanism as recited in claim 42:

wherein said message is multicast using a reliable ordered multicast group communication protocol;

wherein said multicast protocol delivers messages reliably and in the same order to all of the replicas.

44. A software mechanism as recited in claim 43, wherein the mutexes are granted in the same order to the threads in all of the replicas.

45. A software mechanism as recited in claim 44, wherein if two different threads in the replicas both issue a particular claim for a mutex, the message containing said claim that the multicast protocol orders and delivers first determines which claim the operating system's thread library grants first.

46. A software mechanism as recited in claim 39:

wherein if the application program runs on an operating system that provides Dynamically Linked Libraries (DLL), the dynamic linking mechanisms are used to interpose the Consistent Multithreading (CMT) library ahead of the operating system's thread library; and

wherein when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said interpositioning to a function of the CMT library which subsequently invokes a function of the operating system's thread library.

47. A software mechanism as recited in claim 39, further comprising:

adding a statement to the makefile for said application program that causes the linker to interpose the CMT library ahead of the operating system's thread library if the operating system does not provide Dynamically Linked Libraries; and

wherein, when said replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said interpositioning to a function of the CMT library which subsequently invokes a function of the operating system's thread library.

48. A software mechanism for achieving strong replica consistency for replicated multithreaded application programs using an active replication strategy, comprising:

control program code configured to sanitize multithreaded application programs by masking multithreading as a source of non-determinism to ensure that

two or more replicas of a process, object or component maintain the same state.

49. A software mechanism for replicating a multithreaded application program based on an active replication strategy, wherein said application program executes under the control of an operating system having a thread library, the mechanism comprising:

a Consistent Multithreading (CMT) library that is interposed ahead of the operating system's thread library;

wherein said Consistent Multithreading (CMT) library contains wrapper functions for functions of the operating system's thread library;

wherein when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said interpositioning to a wrapper function of the CMT library which subsequently invokes a function of the operating system's thread library.

50. A software mechanism as recited in claim 49, wherein when a replica invokes a function of the Consistent Multithreading (CMT) library to claim a mutex, the Consistent Multithreading (CMT) library function multicasts a message, containing mutex ordering information for that claim, to all of the replicas of the process, object or component and subsequently invokes the corresponding function of the operating system's thread library.

51. A software mechanism as recited in claim 50:

wherein said message is multicast using a reliable ordered multicast group communication protocol;

wherein said multicast protocol delivers messages reliably and in the same order to all of the replicas.

52. A software mechanism as recited in claim 51:

wherein the mutexes are granted in the same order to the threads in all of the replicas.

53. A software mechanism as recited in claim 50, wherein if two different threads each issue a particular claim for a mutex which is multicast in a message, the message containing the claim that the multicast protocol orders and delivers first determines which claim the operating system's thread library grants first.

54. A software mechanism as recited in claim 49:

wherein if the application program runs on an operating system that provides Dynamically Linked Libraries (DLL), the dynamic linking mechanisms are used to interpose the Consistent Multithreading (CMT) library ahead of the operating system's thread library; and

wherein when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said interpositioning to a function of the CMT library which subsequently invokes a function of the operating system's thread library.

55. A software mechanism as recited in claim 49, further comprising:

adding a statement to the makefile for said application program that causes the linker to interpose the CMT library ahead of the operating system's thread library if the operating system does not provide Dynamically Linked Libraries; and

wherein when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said interpositioning to a function of the CMT library which subsequently invokes a function of the operating system's thread library.

56. A software mechanism for replicating a multithreaded application program with an active replication strategy, wherein said application program executes under the control of an operating system, said mechanism comprising:

control program code;

said control program code configured to use a multicast group communication protocol to render a replicated multithreaded application program virtually deterministic.

57. A software mechanism as recited in claim 56:

wherein said control program code is configured to use mutexes to protect shared resources, accessed by threads in the replicas of said replicated multithreaded application program;

wherein said threads are granted the mutexes to access the shared resource in the same order at all of the replicas of said replicated multithreaded application program.

58. A software mechanism as recited in claim 57, wherein said shared resources comprise shared data.

59. A software mechanism as recited in claim 57:

wherein said control program code is configured to intercept calls to the functions of the operating system thread library at each replica;

wherein said control program code is configured to multicast, to the replicas, ordering information regarding the order in which the threads in the replicas claim mutexes.

60. A software mechanism as recited in claim 59, wherein multicast messages that contain ordering information regarding the order in which threads claim mutexes are delivered reliably and in the same order to all of the replicas.

61. A software mechanism as recited in claim 60, wherein if multicast messages from different replicas contain ordering information so that each of two different threads in two different replicas claim the same mutex, then the message that the multicast protocol orders and delivers first determines which claim the operating system's thread library grants first.

62. A software mechanism as recited in claim 61, wherein strong replica consistency and application transparency are maintained by interpositioning a consistent multithreading library ahead of the operating system's thread library and intercepting calls to functions of said operating system's thread library, so as to

render the replicated application program virtually deterministic.

63. A software mechanism as recited in claim 62, wherein functions of the operating system's thread library are wrapped by functions of the consistent multithreading library, and wherein, when a replica of said replicated multithreaded application program invokes a function to claim or release a mutex, said invocation is diverted by said interpositioning to a wrapper function of the CMT library which subsequently invokes the corresponding function of the operating system's thread library.

64. A software mechanism as recited in claim 63, wherein a replica can decide not to multicast information related to the claim of a mutex by a thread in the replica if the replica has already received a message that contains information relating to the same claim of the mutex by the corresponding thread in another replica.

65. A software mechanism as recited in claim 64, wherein if a replica receives a message containing information related to the claim of a mutex by a thread in a replica, and if the replica has already received a message that contains information related to the same claim of the mutex by the corresponding thread in another replica, the replica ignores the information contained in the later received message.

66. A software mechanism as recited in claim 65:

wherein said control program code is configured to allow concurrent processing of threads that do not attempt to claim the same mutex and concurrent processing of threads that claim different mutexes;

wherein strong replica consistency is maintained.

67. A system for executing threads in replicas of an application program within a computing environment, using the active replication strategy, in which resources are shared between threads in a replica, wherein said shared resources

comprise either data or code sections for manipulating said data, comprising:

means for communicating to multiple replicas the claims of shared resources by threads in a replica; and

means for ordering shared resource claims to be granted to threads in multiple replicas corresponding to the order in which claims for the resources were communicated, ordered and delivered through said means for communicating the order of claiming.

68. A system as recited in claim 67, wherein said means for ordering resource claims is configured to prevent threads from being granted a shared resource until claim information has been communicated to the replicas by at least one of said replicas.

69. A system as recited in claim 68, wherein each said replica is configured to order shared resource claims in response to the order of arrival of said shared resource claim information communicated to said replicas.

70. A system as recited in claim 69, wherein said shared resource claim information is communicated by multicasting a message to said replicas.

71. A system as recited in claim 70, wherein shared resource claim granting is configured to maintain an identical claim granting order across said replicas.

72. A system as recited in claim 67, wherein said means for communicating to multiple replicas comprises a means for simultaneously communicating shared resource claim information from a thread in a given replica to all of the replicas.

73. A system as recited in claim 72, wherein said means for simultaneously communicating shared resource claims comprises the multicasting of messages to said replicas.

74. A system as recited in claim 73, wherein said multicast messages contain information about which shared resource is being claimed, which thread is claiming the given shared resource, and which shared resource claim request of said thread is being made.

75. A system as recited in claim 74, wherein said information of which shared resource claim request of said thread is being made comprises a shared resource claim number.

76. A system as recited in claim 67, wherein said means for ordering shared resource claims comprises a routine for selectively granting a resource request, with a particular resource claim number, based on the order in which that resource request was communicated, ordered and delivered to the replicas.

77. A system as recited in claim 67, wherein said means of ordering shared resource claims controls access to a mutual exclusion construct (mutex) through which access to said shared resource is controlled.

78. A system as recited in claim 77, wherein access to said shared resource is controlled by multicasting a message to said replicas.

79. A system as recited in claim 78, wherein said message being multicast comprises elements that identify a shared resource, a thread wanting to access said shared resource, and the thread's resource claim number.

80. A system as recited in claim 67:

wherein said means for communicating and said means of ordering are configured to transparently execute within said system;

wherein said transparent execution is provided in application replicas without modifying the application code.

81. A system as recited in claim 80, wherein said transparent execution comprises executing functions of a consistent multithreading library for said replicas that do not require the application program code to be modified to perform said means for communicating, and said means for ordering.

82. A system as recited in claim 67, further comprising means for communicating pending claims for shared resources to the replicas.

83. A system as recited in claim 82, wherein said means for communicating pending claims for shared resources comprises multicasting a message to all of the replicas.

84. A system as recited in claim 83, wherein said message comprises information about the order in which the threads claim said shared resource.

85. A system as recited in claim 67, wherein said means for ordering shared resource claims in which each corresponding shared resource in a replica maintains identical ordering of corresponding accesses to that shared resource for the corresponding threads in that replica.

86. A system as recited in claim 85, wherein said resource claim information for a particular thread is recorded in an ordered claims queue in the order in which said claims are to be fulfilled.

87. A system as recited in claim 67, wherein said means for communicating, and said means for ordering, are configured as a set of functions of a consistent multithreading library that are executed in response to selected calls to functions of the operating system's thread library.

88. A system as recited in claim 87, wherein said set of functions are configured to intercept calls to select functions within the operating system's thread library.

89. A system as recited in claim 88, wherein said set of functions are dynamically linked into the replicas of said program.

90. A system as recited in claim 89, wherein said set of functions are configured for claiming or releasing a shared resource in a virtually deterministic manner in which identical threads in separate replicas are granted resource access in an identical order.

91. A system as recited in claim 89, wherein said functions of the consistent multithreading library are configured to maintain replica consistency without the need to modify object code, impose single threaded execution, impose distinctions such as Primary and Backup on the replicas, or count instruction execution between non-deterministic events.

92. A system as recited in claim 87, wherein a thread within a replica is suspended upon claiming a shared resource until said claim is communicated to all of said replicas.

93. A system as recited in claim 92, wherein said thread suspension is suppressed if said shared resource claim has already been communicated to all of the replicas.

94. A system as recited in claim 87, wherein said functions of said consistent multithreading library, including said means for communicating, and said means for ordering are configured to maintain strong replica consistency, by forcing virtually deterministic execution.

95. A system as recited in claim 94, wherein the threads in said replicas are forced to access the resources that they share in an identical order.

96. A system as recited in claim 94, wherein said strong replica consistency is maintained within client-server systems.

97. A system as recited in claim 94, wherein said strong replica consistency is maintained within fault-tolerant systems.

98. A system as recited in claim 94, wherein for said replicas strong replica consistency is maintained without distinguishing replicas into roles as either Primary or Backup replicas.

99. A system as recited in claim 94, wherein said strong replica consistency is maintained without the need to provide special hardware.

100. A system as recited in claim 94, wherein said strong replica consistency is maintained without the need to perform hardware or software instruction counting between non-deterministic events.

101. A system as recited in claim 94, wherein said strong replica consistency is maintained without modifying either the application source code or the application object code.

102. A system as recited in claim 87, wherein said shared resource comprises memory or an input/output location accessible to the threads of said replica.

103. A system as recited in claim 87, wherein said means for communicating is configured to communicate messages containing shared resource claim information.

104. A system for executing threads in replicas within a computing environment adapted for executing program replicas, based on an active replication strategy comprising:

an allocation routine configured for granting access to shared resources to threads in said replica, in response to information delivered about the order in which threads in said and other replicas claim access to shared resources; and

means for communicating the order in which threads in said replica claim access to shared resources, to all of the replicas within said system.

105. A system as recited in claim 104, wherein said means for communicating the order of accessing shared resources comprises a routine configured for multicasting messages to multiple replicas in response to access requests and selected state changes.

106. A consistent multithreading library of thread functions for constraining access to shared resources by threads in a replica within a computing environment configured for executing multiple program replicas, based on the active replication strategy, comprising functions that invoke:

a communication routine configured for communicating allocation information about said shared resources to multiple replicas within said computing environment; and

an allocation routine configured for allocating shared resources to threads in a replica in response to information delivered, related to the order in which said shared resources were claimed by the threads in the replicas.

107. A library as recited in claim 106:

wherein said library of thread functions wrap functions of the operating system's thread library;

wherein said library is interposed ahead of the operating system's thread library.

108. A library as recited in claim 107:

wherein said wrapper functions include programming configured for carrying out the operations of

invoking communication and allocation routines, and

invoking functions of said operating system's thread library.

109. A library as recited in claim 107, wherein said interception of calls to the operating system's thread library is performed by a dynamic linking process.

110. A library as recited in claim 109, wherein said interception of calls to the operating system's thread library is performed by inserting commands in the makefile for said program, to cause the linker to position said library of thread functions ahead of the operating system's thread library.

111. A library as recited in claim 109, wherein said library of thread functions are configured as a set of functions incorporated within the operating system's thread library.

112. An apparatus for executing threads in replicas within a computing environment, using the active replication strategy, while maintaining strong replica consistency across said replicas, comprising:

a computer configured for executing said replicas; and

programming associated with said computer for carrying out the operations of

communicating to multiple replicas the order of requests for access to shared resources by threads in a replica, and

ordering the granting of resource access requests by the threads in a replica in response to the order in which said resource access requests are ordered and delivered.

113. A media that is computer readable and includes a computer program which, when executed on a computer configured for multithreaded execution and communication with multiple program replicas, causes the computer to execute instructions, comprising:

communicating to multiple replicas the order of claiming of shared resources by threads in a replica; and

ordering the granting of resource claims to corresponding threads in multiple replicas in response to the order in which the claims of said threads for access to the shared resources are ordered and delivered to the multiple replicas.

114. A system executing a multithreaded application program within a computing environment in which said program is replicated using the active replication strategy, comprising:

a consistent multithreading library having functions configured for suspending a thread that claims a shared resource;

a multicast group communication protocol for communicating the order of the claims of the shared resources by the threads in the replicas; and

resource granting functions configured for granting access to shared resources, and activating suspended threads, in response to said ordering of shared resource claims when said resources are available.

115. A system as recited in claim 114, wherein a resource identifier, thread identifier, and claim identifier, are communicated by said multicast group communication protocol.

116. A system as recited in claim 114, wherein said replicated multithreaded program is rendered virtually deterministic in order to maintain strong replica consistency.

117. A system as recited in claim 114, wherein said resource granting functions are executed in conjunction with a consistent multithreading library for controlling access to shared resources.

118. A system as recited in claim 117:

wherein said functions of said consistent multithreading library intercept calls to functions of the operating system's thread library,

wherein strong replica consistency is maintained while said functions of the consistent multithreading library remain transparent to said application program.

119. A system as recited in claim 114, wherein the order in which said multicast protocol delivers messages containing a claim by a given thread for access to a given shared resource for a given claim number, determines the order in which

the claims to the shared resource are granted to said threads in said replicas within said computing environment.

120. A system as recited in claim 114, wherein said thread suspension, multicast group communication of claim order, and resource granting functions are invoked by functions of the consistent multithreading library that intercept the functions for claiming and releasing resources provided by the operating system's thread library.

121. A system as recited in claim 114, wherein multicasting of a claim request can be suppressed if a given replica has already received the same claim for the same shared resource by the same thread.

122. A system as recited in claim 114, wherein threads that are not currently claiming the same shared resources are allowed to process concurrently.

123. A system as recited in claim 114, wherein said communication comprises the multicasting of messages containing information, or the writing into shared memory of information, related to the claiming of shared resources.

124. A system for replicating multithreaded application programs on computers within a computing environment, using the active replication strategy, comprising:

a consistent multithreading library configured for linking with said application program and processing shared resource access requests; and

a means for maintaining an identical order of shared resource access by threads within replicas, thereby providing virtual determinism and strong replica consistency.

125. A system as recited in claim 124, wherein said computers that host said replicas of said application programs may provide multithreading, multitasking, distributed computing, fault tolerance, a client-server paradigm, and combinations

thereof.

126. A system as recited in claim 124, wherein said means for maintaining an identical order of resource access comprises functions of a consistent multithreading library that are executed in response to shared resource access requests and associated operations.

127. A system as recited in claim 126, wherein said functions of said consistent multithreading library are configured as wrapper functions of the functions of the operating system's thread library.

128. A system as recited in claim 126, wherein said wrapper functions of consistent multithreading library intercept calls to the operating system's thread library and execute preceding the execution of functions of the operating system's thread library.

129. A system as recited in claim 128, wherein said consistent multithreading library is configured for being dynamically linked into said application programs.

130. A system as recited in claim 127, wherein said functions of said consistent multithreading library are configured to:

communicate shared resource claim information to all of the replicas;

order resource claim information in each replica in response to the order in which shared resource claims are delivered; and

grant shared resource claims in each replica in response to said order, wherein the corresponding threads of the replicas access corresponding shared resources in an identical order.

131. A system as recited in claim 130, wherein said library is configured for maintaining pending claims and ordered claims queues that record the order in which mutexes are claimed, granted and released.

132. A system as recited in claim 124, wherein said means for maintaining an identical order of resource access comprises:

multicasting a message identifying resource claims; and

granting claims for resources in response to the order in which messages identifying resource claims are delivered.

133. A system as recited in claim 132, wherein said message contains information that identifies the shared resource for which a claim is being made, the thread that is claiming the resource, and the thread's resource claim number.

134. A system as recited in claim 133:

wherein said granting of claims, in response to the order in which messages containing resource claims are delivered, causes the suspension of threads claiming a shared resource, until said resource is available and any prior claims for that resource have been granted;

wherein a prior claim for a resource, relative to a specific claim, is a claim contained in a message that is ordered and delivered ahead of the first message that contained said specific claim.

135. A method of maintaining virtually deterministic behavior among multiple replicas of a replicated multithreaded application program where corresponding threads in the replicas access corresponding shared resources, comprising:

communicating requests for access to shared resources to all of the replicas of said application program; and

granting access requests for shared resources to threads in said replicas in response to the order in which requests for access to said shared resources are delivered to the replicas.

136. A method as recited in claim 135, wherein the order in which a thread in a replica of said application program accesses a shared resource is identical across all of the replicas to provide virtually deterministic behavior.

137. A method as recited in claim 135, wherein said granting of access requests for shared resources comprise functions of a consistent multithreading library.

138. A method as recited in claim 137, wherein said consistent multithreading library is interposed ahead of the operating system's thread library and where the functions of said consistent multithreading library wrap the corresponding functions of the operating system's thread library.

139. A method as recited in claim 138, wherein said consistent multithreading library is interposed ahead of the operating system's thread library by dynamic or static linking.

140. A method as recited in claim 139, wherein a claim for a resource by a thread that was ordered and delivered determines the access order for the corresponding shared resources by corresponding threads in all of the replicas.

141. A method as recited in claim 140, wherein said thread is suspended after claiming a resource until said request has been communicated, ordered and delivered.

142. A method of executing actively replicated multithreaded application programs containing threads that access shared resources, comprising:

invoking a claim for a shared resource by a thread;

multicasting a message to communicate said shared resource claim to all of the replicas of said application program;

ordering of shared resource claim requests; and

granting said shared resource to a given thread in response to the order in which multicast messages are communicated, ordered and delivered, and to the availability of said shared resource.

143. A method as recited in claim 142, further comprising releasing said shared resource by said thread allowing other threads to access said shared resource.

144. A method as recited in claim 142, further comprising suspending said thread making said shared resource claim until after said claim information has been communicated, ordered and delivered to the replicas.

145. A method as recited in claim 142, wherein said multicast message contains information regarding the shared resource being claimed, the thread claiming the shared resource, and the thread's claim request number.

146. A method as recited in claim 145, wherein said information about claim requests is recorded in a pending claims queue.

147. A method of identically ordering accesses to corresponding shared resources by corresponding threads in different replicas of an actively replicated multithreaded application program, comprising:

invoking a claim on a shared resource by a thread in a replica in preparation for accessing said shared resource;

suspending execution of said thread invoking said resource claim;

multicasting a message to communicate information about said resource claim by said thread in said replica to all of the replicas;

ordering and delivering said information about said resource claims by said multicast;

granting access to said resource, when available, to a thread in response to the order in which said information is delivered to the replicas; and

releasing said resource, by said thread granted said resource, after said thread has completed accessing said shared resource.

148. A method as recited in claim 147, wherein access to said shared resource is controlled by a mutual exclusion construct (mutex), to which a claim by a

given thread must be granted prior to that thread accessing said shared resource associated with said mutual exclusion construct.

149. A method as recited in claim 148, wherein information about said shared resource claims is recorded in pending claims and ordered claims queues.

150. A method as recited in claim 148, wherein said claim to a mutual exclusion construct comprises:

determining *(T, M, N)* for a claim to a mutual exclusion construct associated with said shared resource, wherein *T* represents the thread making said claim, *M* represents the mutual exclusion construct being claimed, and *N* represents the claim number by thread *T* to any mutex;

determining if *(T, N)* is the next thread and claim number for which access to the mutual exclusion construct *M* is to be granted, and if so,

(i) recording that thread *T* has been granted its *N*th claim,

(ii) marking said mutual exclusion construct as unavailable,

(iii) granting said mutual exclusion construct to thread *T*, and

(iv) bypassing following steps (c) through (e);

determining that a claim *(T, N)* has not been delivered, in which case information about the claim is multicast to other replicas;

marking thread *T* claiming mutual exclusion construct *M* as suspended; and

suspending thread *T*.

151. A method as recited in claim 150, wherein said resource releasing comprises:

marking the mutual exclusion construct associated with said resource as available;

selecting the next ordered and delivered but ungranted claim *(T', N')* for the mutual exclusion construct, wherein *T'* represents the thread making said resource claim and *N'* represents the claim number by thread *T'*; and

signaling thread *T'* to awaken thread *T'*, if thread *T'* is suspended.

152. A method as recited in claim 147, wherein said method of identically ordering accesses to shared resources by threads in replicas comprises a consistent multithreading library containing functions that control access to shared resources.

153. A method as recited in claim 152, wherein said consistent multithreading library is interposed ahead of the operating system's thread library and where the functions of the consistent multithreading library wrap the corresponding functions of the operating system's thread library.

154. A method as recited in claim 153, wherein said consistent multithreading library is interposed ahead of the operating system's thread library by dynamic or static linking.

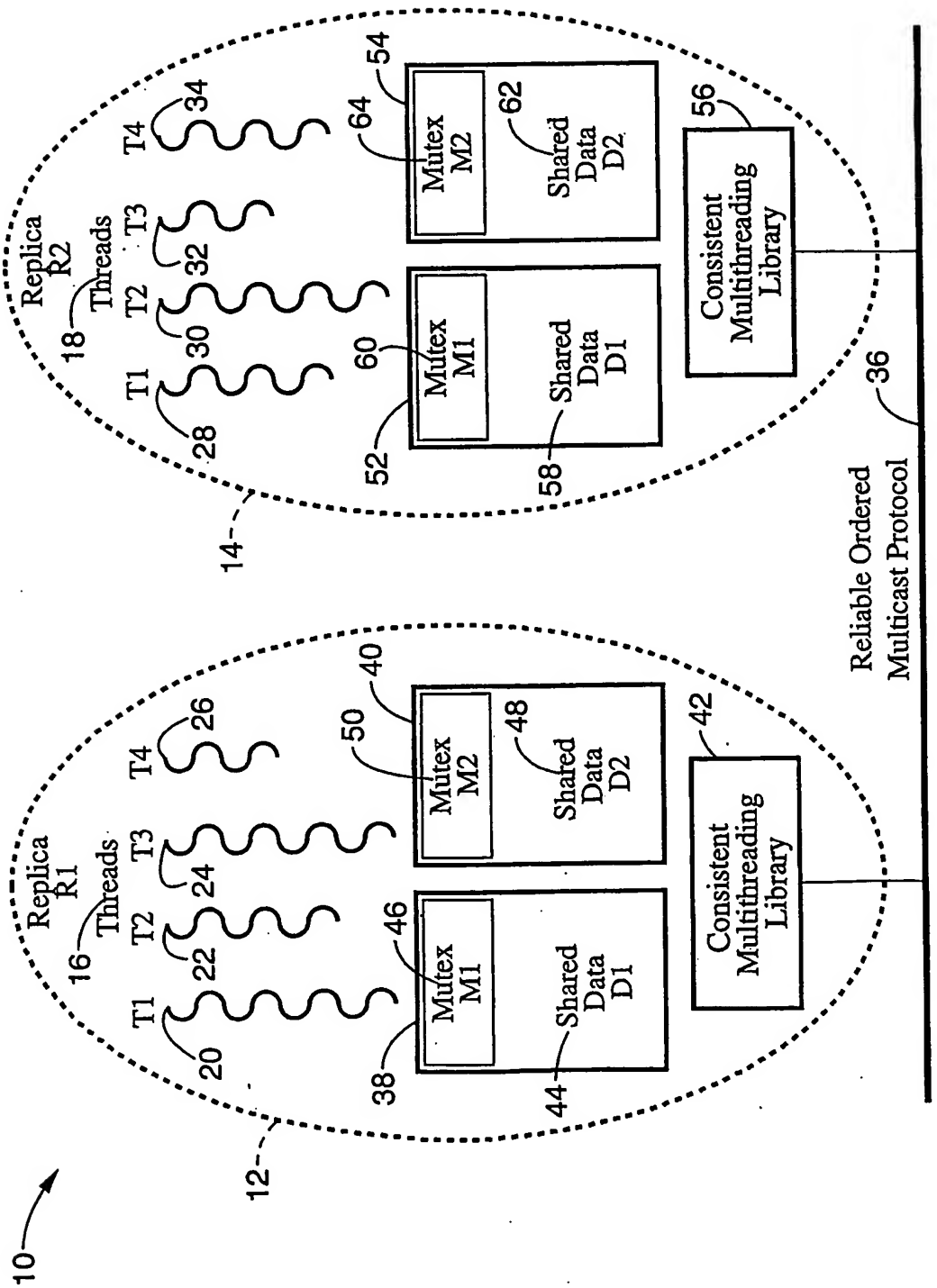

10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
48
50
52
54
56
58
60
62
64
Replica R1
Threads
T1
T2
T3
T4
Mutex M1
Shared Data D1
Mutex M2
Shared Data D2
Consistent Multithreading Library
Replica R2
Threads
T1
T2
T3
T4
Mutex M1
Shared Data D1
Mutex M2
Shared Data D2
Consistent Multithreading Library
Reliable Ordered Multicast Protocol

FIG. 1

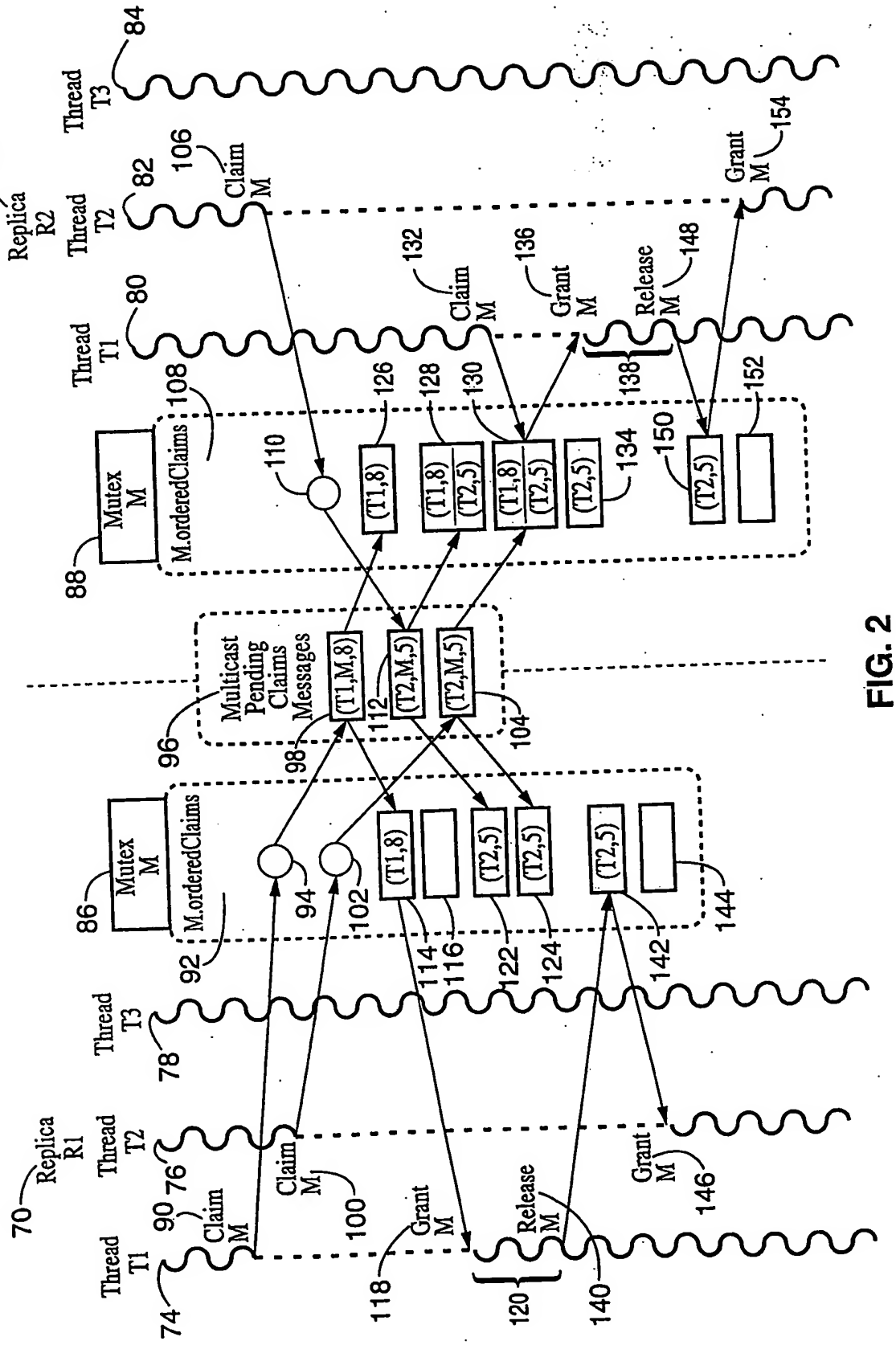

Replica R1
Replica R2
Thread T1
Thread T2
Thread T3
Mutex M
M.orderedClaims
Multicast Pending Claims Messages
Claim M
Grant M
Release M
(T1,8)
(T2,5)
(T1,M,8)
(T2,M,5)
70
72
74
76
78
80
82
84
86
88
90
92
94
96
98
100
102
104
106
108
110
112
114
116
118
120
122
124
126
128
130
132
134
136
138
140
142
144
146
148
150
152
154

FIG. 2

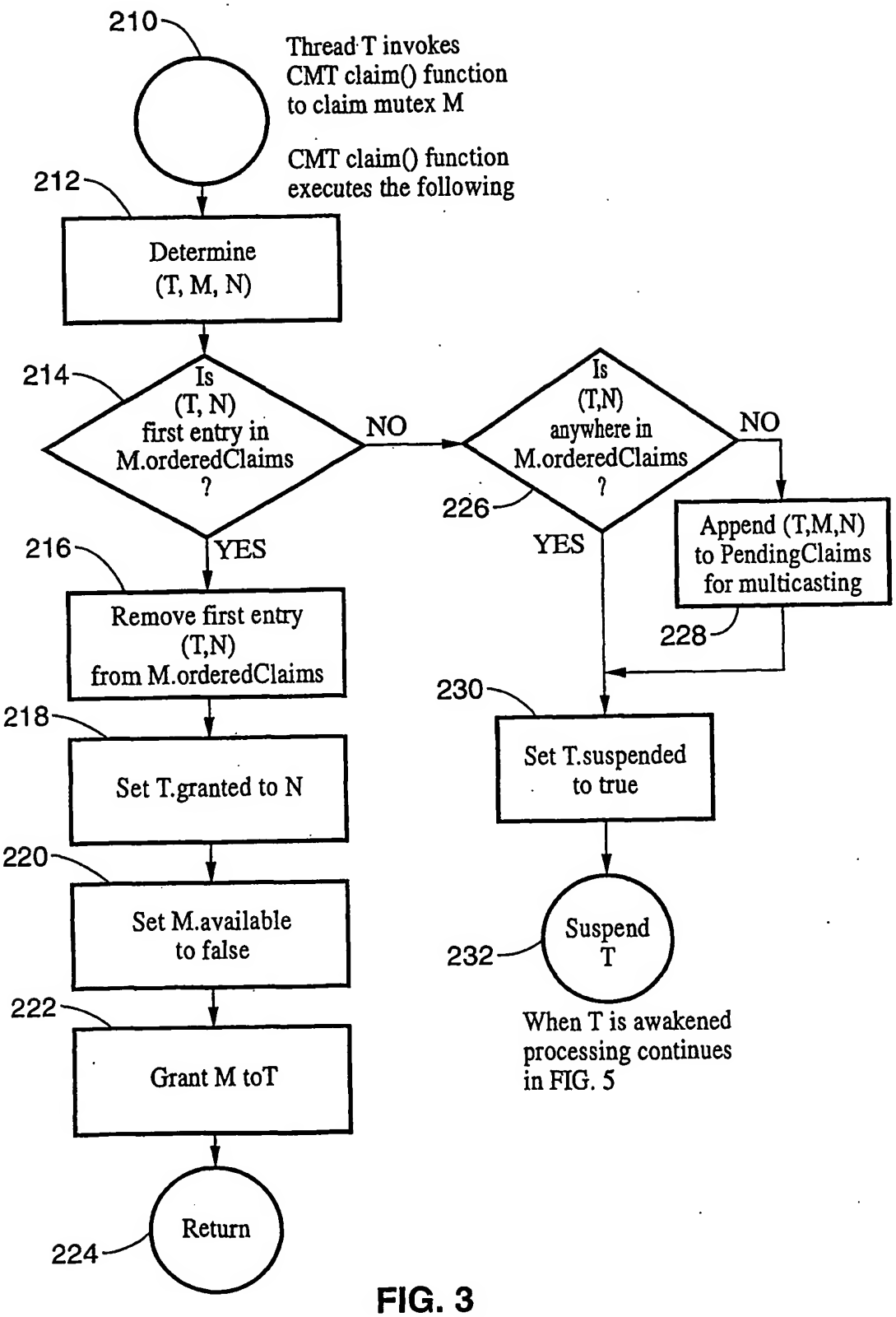

210
Thread T invokes CMT claim() function to claim mutex M
CMT claim() function executes the following
212
Determine (T, M, N)
214
Is (T, N) first entry in M.orderedClaims ?
NO
Is (T,N) anywhere in M.orderedClaims ?
226
NO
YES
YES
216
Remove first entry (T,N) from M.orderedClaims
Append (T,M,N) to PendingClaims for multicasting
228
230
218
Set T.granted to N
Set T.suspended to true
220
Set M.available to false
232
Suspend T
222
Grant M toT
When T is awakened processing continues in FIG. 5
Return
224

FIG. 3

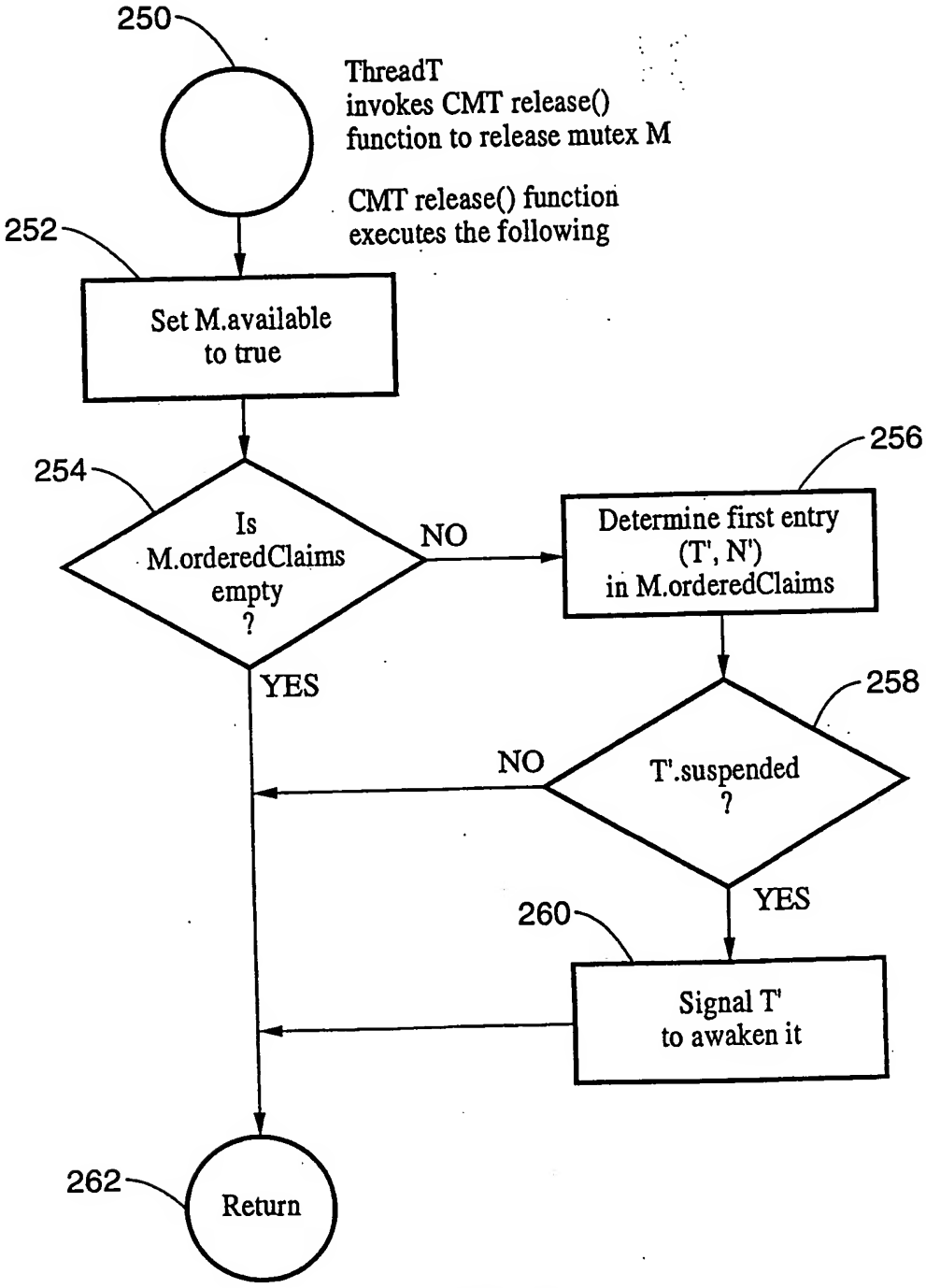

250
ThreadT
invokes CMT release()
function to release mutex M
CMT release() function
executes the following
252
Set M.available
to true
254
Is
M.orderedClaims
empty
?
NO
256
Determine first entry
(T', N')
in M.orderedClaims
YES
258
NO
T'.suspended
?
YES
260
Signal T'
to awaken it
262
Return

FIG. 4

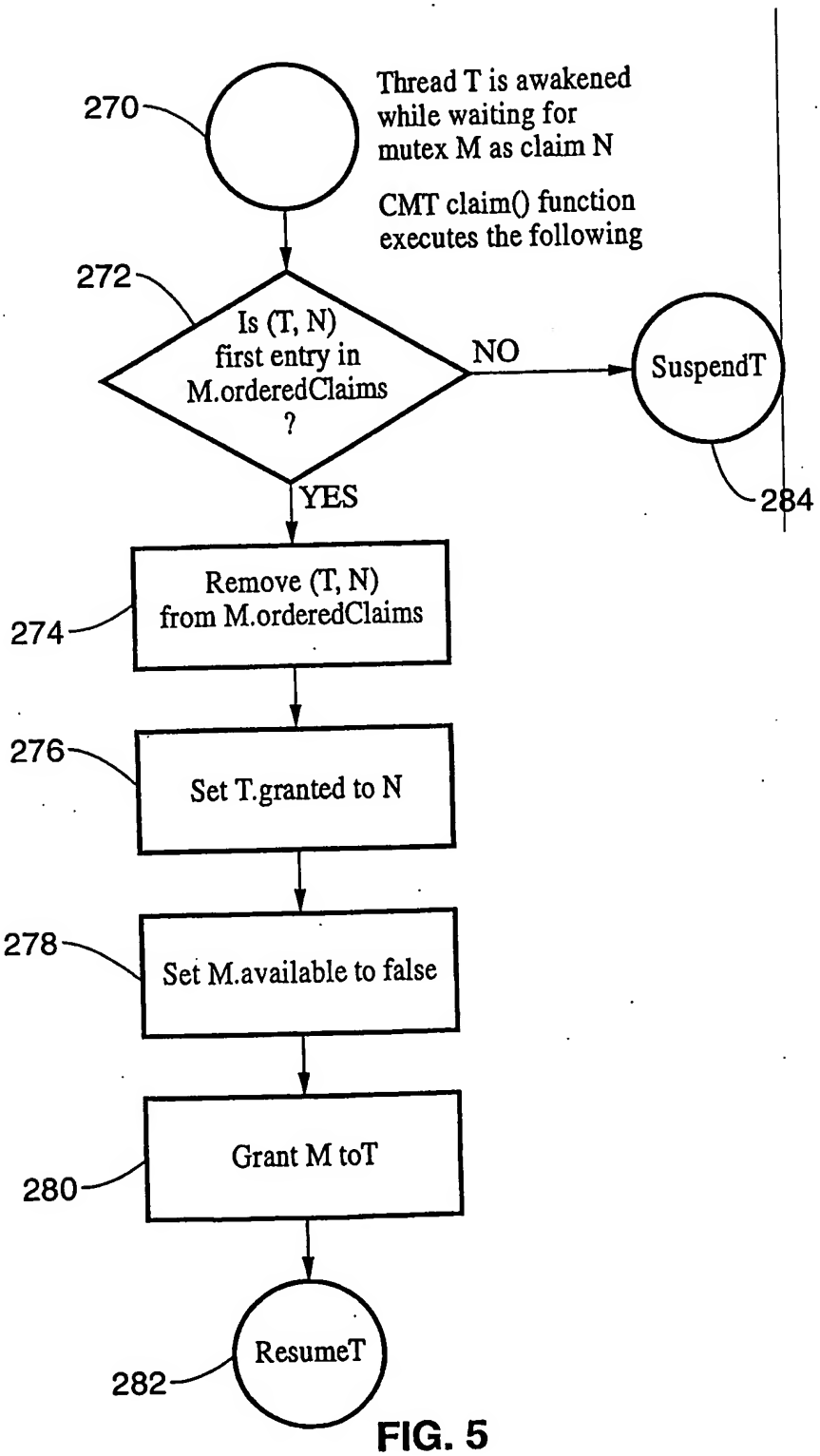

270
Thread T is awakened while waiting for mutex M as claim N
CMT claim() function executes the following
272
Is (T, N) first entry in M.orderedClaims ?
NO
SuspendT
284
YES
Remove (T, N) from M.orderedClaims
274
Set T.granted to N
276
Set M.available to false
278
Grant M toT
280
ResumeT
282

FIG. 5

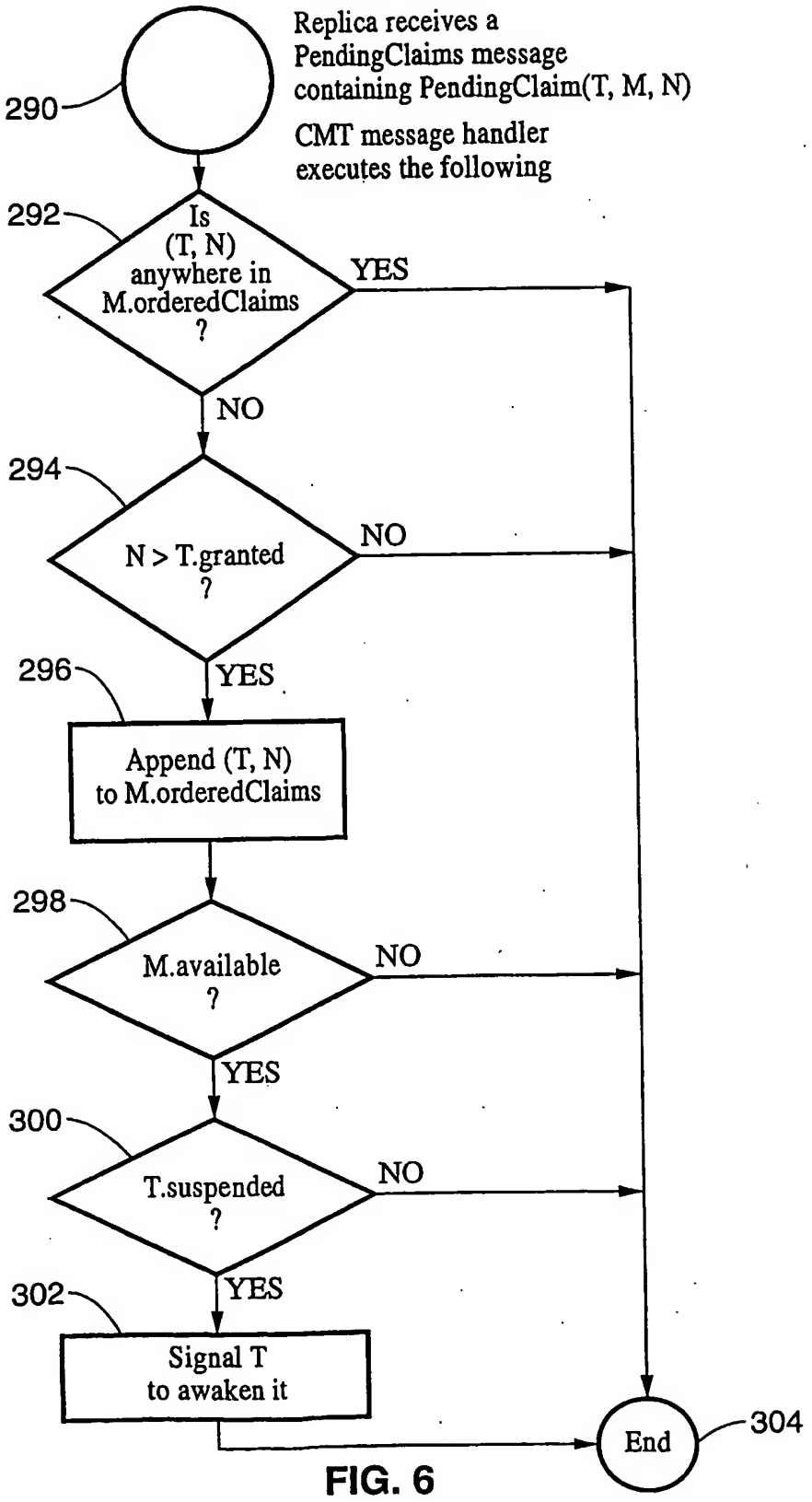

Replica receives a PendingClaims message containing PendingClaim(T, M, N)
CMT message handler executes the following
290
292
Is (T, N) anywhere in M.orderedClaims ?
YES
NO
294
N > T.granted ?
NO
YES
296
Append (T, N) to M.orderedClaims
298
M.available ?
NO
YES
300
T.suspended ?
NO
YES
302
Signal T to awaken it
End
304

FIG. 6